useState()

Connect with Me:
LinkedIn : https://www.linkedin.com/in/priya-bagde
GitHub : https://github.com/priya42bagde
LeetCode : https://leetcode.com/priya42bagde/
YouTube Channel : https://youtube.com/channel/UCK1_Op30_pZ1zBs9l3HNyBw (Priya Frontend Vlogz)

- The **state** is **data or properties** which are **mutable**, meaning **their value can change**, using the useState() hook.
- The useState() hook used to add a **state management(**to handle, create, update and manage your states) **in functional components** and **never use it within a nested function, loop or condition**.
- If you have **complex state, then storing multiple values** in useState can get difficult, then the **useReducer hook** which is better suited to managing state with multiple values.

- ✓ If the **new value you provide is same to the current state**, then React will **skip re-rendering the component and its children.**
- ✓ If you use the **previous value to update state**, you must **pass a function that receives the previous value and returns an updated value**, for example, setMessage(previousVal => previousVal + currentVal)
- ✓ The **state updates are asynchronous (i.e, doesn't** immediately trigger a re-render for each individual state update) and **batched** and **we can use multiple state variables in a component by calling useState hook multiple times**. Also, **React batches multiple setState calls together for a single render**. This means that if you have multiple state updates within the same render cycle, **React will optimize and re-render the component only once**.
- ✓ The useState **does not automatically merge update objects(**it do shallow merge**)**, so we should **replace the state rather than mutate your existing objects** using **spread operator**.

```
const [user, setUser] = useState({ name: '', age: 0 });

// Correct way to update a property inside the user object
setUser((prevUser) => ({ ...prevUser, age: prevUser.age + 1 }));

```

Syntax: `const [count, setCount] = useState(initialState);`

- It returns **an array consisting of two elements**: the **current state and a function** to update the state.
- **The first(initial) time** the component is rendered, **the initial state is passed as the argument to useState**. It's **not a mandatory to initiate the state with initial value. it can be a empty useState function**. But initiating the state with initial value will be the part of good practice. In **class components, the state was always an object**, and you could store multiple values in that object. But with hooks, the state can be any type you want as **an array, object, a number, a boolean, a string, whatever you need**.

Example:

```
import { useState } from 'react'

function App() {
  const [state, setState] = useState(1)

  return (
    <section>
      <div>{state}</div>
      <button onClick={() => setState(state + 1)}>More</button>
      <button onClick={() => setState(state - 1)}>Less</button>
    </section>
  )
}
```

Key Point Explanations:

Updating state based on the previous state

Suppose the `age` is `42`. This handler calls `setAge(age + 1)` three times:

```
function handleClick() {
  setAge(age + 1); // setAge(42 + 1)
  setAge(age + 1); // setAge(42 + 1)
  setAge(age + 1); // setAge(42 + 1)
}
```

However, after one click, `age` will only be `43` rather than `45`! This is because calling the `set` function does not update the `age` state variable in the already running code. So each `setAge(age + 1)` call becomes `setAge(43)`.

To solve this problem, **you may pass an *updater function*** to `setAge` instead of the next state:

```
function handleClick() {
  setAge(a => a + 1); // setAge(42 => 43)
  setAge(a => a + 1); // setAge(43 => 44)
  setAge(a => a + 1); // setAge(44 => 45)
}
```

Here, `a => a + 1` is your updater function. It takes the pending state and calculates the next state from it.

React puts your updater functions in a queue. Then, during the next render, it will call them in the same order:

I've updated the state, but logging gives me the old value

Calling the `set` function **does not change state in the running code**:

```
function handleClick() {
  console.log(count);  // 0

  setCount(count + 1); // Request a re-render with 1
  console.log(count);  // Still 0!

  setTimeout(() => {
    console.log(count); // Also 0!
  }, 5000);
}
```

This is because states behaves like a snapshot. Updating state requests another render with the new state value, but does not affect the `count` JavaScript variable in your already-running event handler.

If you need to use the next state, you can save it in a variable before passing it to the `set` function:

```
const nextCount = count + 1;
setCount(nextCount);

console.log(count);     // 0
console.log(nextCount); // 1
```

Avoiding recreating the initial state

React saves the initial state once and ignores it on the next renders.

```
function TodoList() {
  const [todos, setTodos] = useState(createInitialTodos());
  // ...
```

Although the result of `createInitialTodos()` is only used for the initial render, you're still calling this function on every render. This can be wasteful if it's creating large arrays or performing expensive calculations.

To solve this, you may **pass it as an *initializer* function** to `useState` instead:

```
function TodoList() {
  const [todos, setTodos] = useState(createInitialTodos);
  // ...
```

Notice that you're passing `createInitialTodos`, which is the *function itself*, and not `createInitialTodos()`, which is the result of calling it. If you pass a function to `useState`, React will only call it during initialization.

I've updated the state, but the screen doesn't update

React will **ignore your update if the next state is equal to the previous state,** as determined by an `Object.is` comparison. This usually happens when you change an object or an array in state directly:

```
obj.x = 10;  // 🚩 Wrong: mutating existing object
setObj(obj); // 🚩 Doesn't do anything
```

You mutated an existing `obj` object and passed it back to `setObj`, so React ignored the update. To fix this, you need to ensure that you're always *replacing* objects and arrays in state instead of *mutating* them:

```
// ✅ Correct: creating a new object
setObj({
  ...obj,
  x: 10
});
```